Early Load Address Resolution Via Register Tracking

Michael Bekerman[1], Adi Yoaz, Freddy Gabbay[2], Stephan Jourdan,
Maxim Kalaev and Ronny Ronen

Microprocessor Research Lab
Intel Israel Architecture Research

ISCA-2K June 14, 2000

1. Currently at HAL Computer Systems
2. Currently at Mellanox Technologies Inc

Agenda

- Motivation
- Stack tracking
  - Concept
  - Implementation
  - Performance Results
- Extension to full register tracking

Beyond Data Flow

- Advanced Microarchitecture research attempts to break the data flow barrier:
  - Value prediction [Lipasti '96, Sazeides '97, Gabbay '97]
  - Address Prediction [Gonzales '97, Bekerman '99]
  - Memory Cloaking / Disambiguation
    [Austin '97, Moshovos '97, Chrysos '98, Yoaz '99]
  - Identity Re-Direction [Jourdan '98]
  - Instruction Reuse [Sodani '97]*
- Many of these techniques involve speculative work:
  - Requires more resources
  - Costs power
  - Costs recovery logic
  - ...
- We introduce a **non-speculative** technique:

Register Tracking

* Safe (non speculative) Technique

Motivation: the IA32 Stack

- IA32 (X86) features a HW stack
  - Special Stack Pointer register (ESP)
  - Special Stack operations: *Push, Pop, Call, Ret*
- Every Stack operation generates an ESP updating micro-operation (uop)
- Applications use *stack* extensively for storing and accessing:
  - Procedure parameters and return address
  - Local variables
- **14%** of the dynamic instructions include an ESP updating micro-operation (uop)
- **8%** of the dynamic uop stream modify the ESP value
- **25%** of the loads are ESP based

IA32 Instructions	Operation
PUSH EAX	mem[ESP-4] ← *EAX* ESP ← ESP - 4
POP EAX	*EAX* ← mem[ESP] ESP ← ESP + 4
RETurn **(Pop & Jump)**	*tmp* ← mem[ESP] ESP ← ESP + 4 jump *tmp*
CALL foo **(Push & Jump)**	mem[ESP-4] ← next-IP ESP ← ESP - 4 Jump *foo*
Stack Load	*EAX* ← mem[ESP+imm]

Motivating Example (X86)

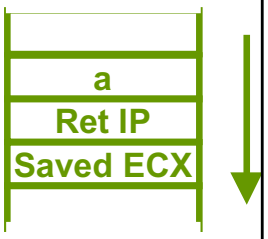
Original C code

```
call foo (a)
...
void foo(int a)
  { … = a; …
  }
```

- Observations:
  - Inter-dependencies are eliminated: More operations can be done in parallel
  - Load/Store addresses are known earlier
  - Load/Store collisions are known earlier
  - Many operations become practically redundant

X86 assembly code

```
push EAX
call  foo
push ECX / save reg
load  [ESP+8], ECX
...
pop   ECX / restore reg
```



Micro Operations

		DF Height
st	EAX, [ESP-4]	1
sub	ESP,4	1
st	NIP, [ESP-4]	2
sub	ESP,4	2
jmp	foo	1
st	ECX, [ESP-4]	3
sub	ESP,4	3
ld	[ESP+8], ECX	4
…		
ld	[ESP], ECX	4
add	ESP,4	4

But if we track ESP

		DF Height
st	EAX, [ESP_0-4]	1
sub	ESP_0, 4, ESP	X
st	NIP, [ESP_0-8]	1
sub	ESP_0,8, ESP	X
jmp	foo	1
st	ECX, [ESP_0-12]	1
sub	ESP_0,12, ESP	X
ld	[ESP_0-4], ECX	2
…		
ld	[ESP_0-12], ECX	2
add	ESP_0,-8, ESP	X

Register / Stack Tracking

MRL

A non-speculative technique that:

- Increases ILP by collapsing dependencies
- Allows earlier knowledge of load addresses: Reduces the load-to-use latency
- Enables earlier resolution of memory ambiguity
- Reduces the stream of micro-operations (“Uops”)

intel

Stack Tracking *Concept overview*

Part-I: ESP Value Tracking

- Goal: latest ESP value available at the Front-End (FE)
- Method: Track ESP changes in FE
  - Trackable uops are simple ESP modifications of the form:
    - *ESP ESP± immediate*
    - *ESP immediate* (quite rare)
    - Other ESP-updating uops are untrackable (*load into ESP, Move Reg into ESP,..)*
  - ➡ Enable decode-time information of register values
- *"Execute"* Trackable ESP-modifying uops in the FE
  - Eliminate them from flowing at the rest of the pipeline
- ➡ ***Cut dependencies, Reduce Uop Stream***
- J >98% modifications to ESP are trackable

Stack Tracking *Concept overview (Cont.)*

MRL

Part-II: Early Load Resolution

- Compute the addresses of ESP-based Loads in the Front-End using the tracked ESP value

EAX mem[ESP ± *immediate*]

- Apply early resolved addresses to the disambiguation mechanism
- Issue early resolved Loads to memory in earlier pipeline stages to hide L1 cache latency
  - Practically reducing load-to-use delays

➡ ***Collapse program critical path and improve ILP***

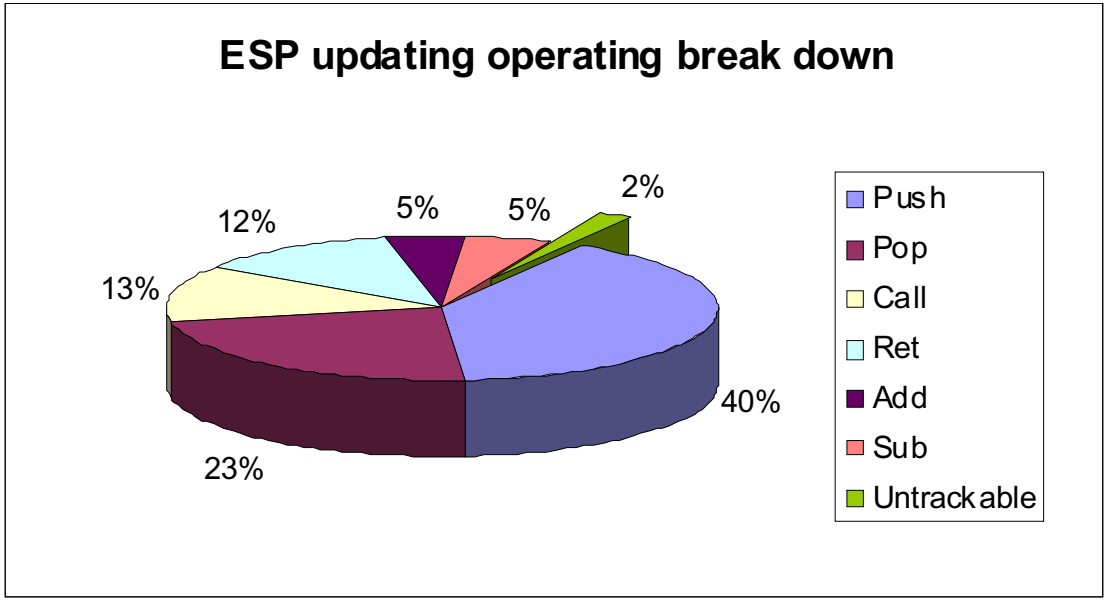
J >97% of references to the stack are ESP based*

*** Code styles differ: old code uses EBP =>**
more EBP based memory references and LEAVE instruction

intel

ESP Trackability

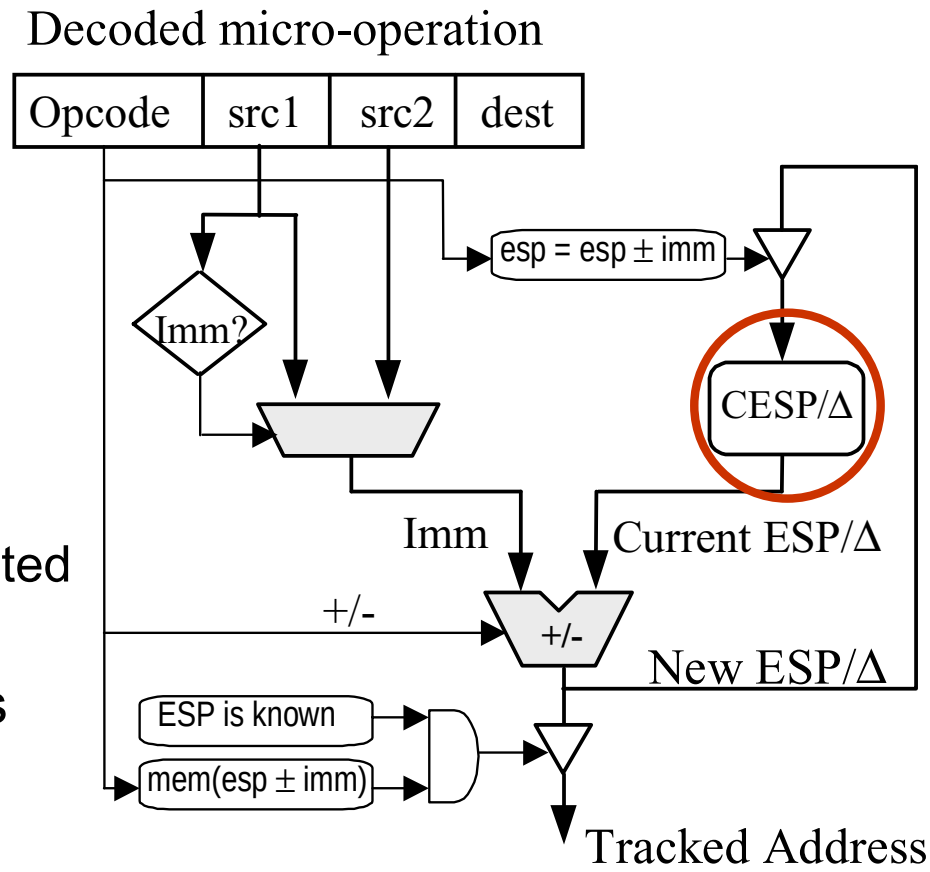
- **Most ESP updating operations are Trackable:**
  - 98% of the IA's that change the ESP are trackable:
    - 40% of the changes come from PUSHs
    - 23% come from POPs,
    - 13% are CALLs, 12% are RETs,
    - 5% are ADDs, 5% are SUBs
  - 2% of the IA's that changes the ESP are untrackable
    - Mostly ESP Reg, ESP ESP±REG, and loads to ESP



ESP Tracking HW

- CESP (Current ESP) records the current value of ESP in the FE
- Simple ESP modifications use the CESP value and update it
- When the (Current) ESP is known
  - Memory references that use ESP receive an early computed address
  - Simple ESP modifications are executed in the Front-End (uop-elimination)
- Multiple ESP updates and retrievals can be accomplished in a cycle
- An untrackable ESP-updating uop
  - Invalidates the CESP value: making it *UNKNOWN* till the uop is resolved
  - Passes its computed value (after execution) to the tracking HW

Longer pipeline Ł longer *UNKNOWN* period



ESP Tracking HW (con't)

- When CESP is unknown, an ESP Δ is computed
  - When the untrackable ESP modifying uop is finally computed, CESP value is reconstructed by summing the Δ and the computed ESP
  - Several untrackable ESP uops may be in flight - Δ refers to the last in order
- During the time the (Current) ESP is unknown
  - Memory references that use ESP cannot receive an early computed address
  - Simple ESP-modifying uops should be fully executed
- CESP value must be recovered following Branch mispredictions or Interrupts

- Important: distinguish between "Trackable" and "Known"
  - Trackable/Untrackable ESP modifying uop: Static attribute - depends on the uop only
  - Known/Unknown ESP uop: Dynamic attribute - depends on pipeline length
  - Known uops $\subseteq$ Trackable uops

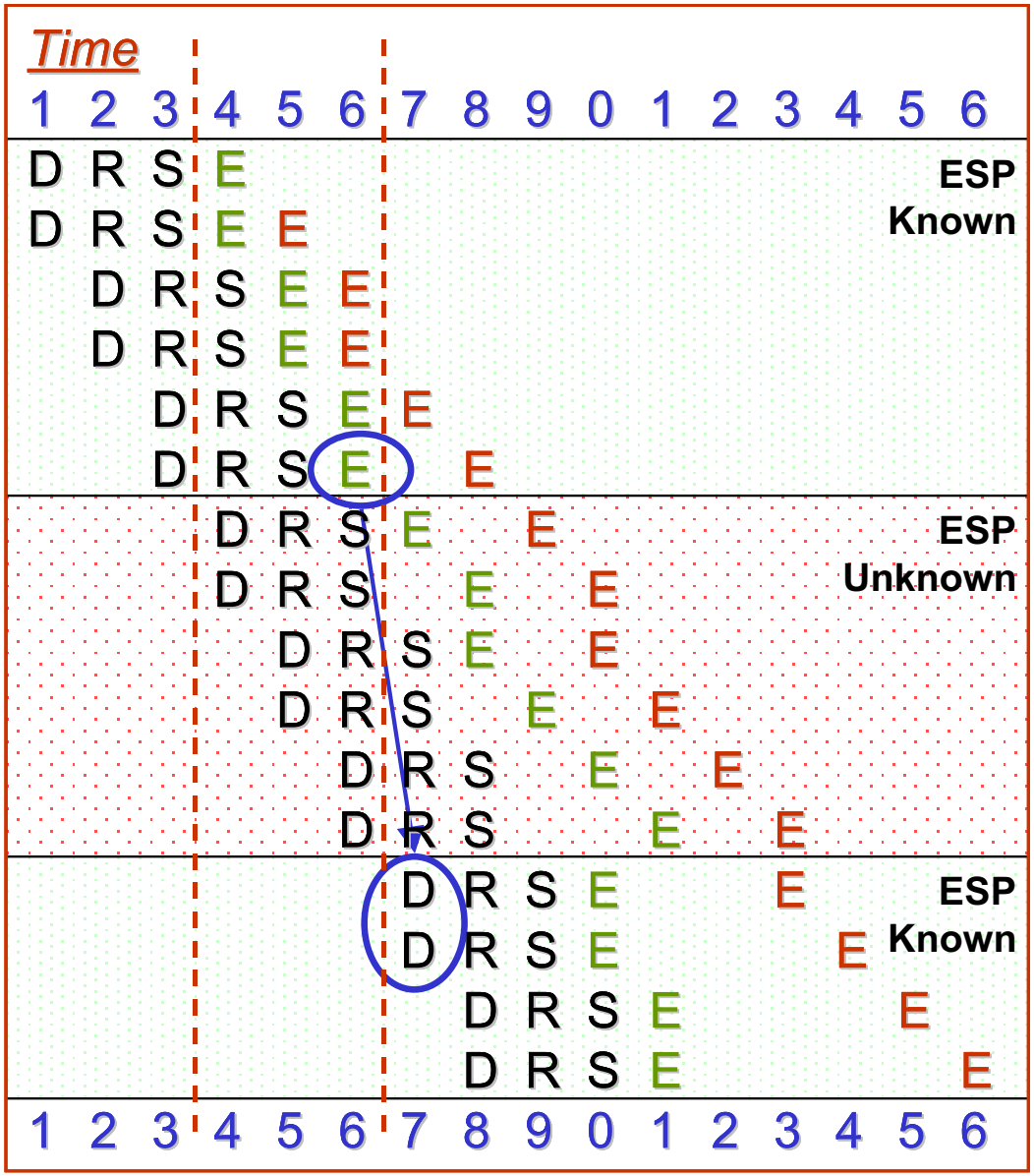
ESP Tracking Example

MRL

Micro Operations

T_0: *ESP = 24*

1. add ESP, 4
2. add ESP, 4
3. load [ESP+4], EAX
3. add ESP, 4
4. add ESP, 4
5. and ESP, 0xf
6. add ESP, 4
8. load [ESP+8], EBX
9. add ESP, 4
10 add ESP, 4
11 add ESP, 4
12 load [ESP+2], ECX
13 add ESP, 4
14 add ESP, 4
15 add ESP, 4
16 load [ESP+6], EDX



CESP / Δ

24
28
2C
2C
30
34
0
4
4
8
C
10
14
18
1C
20

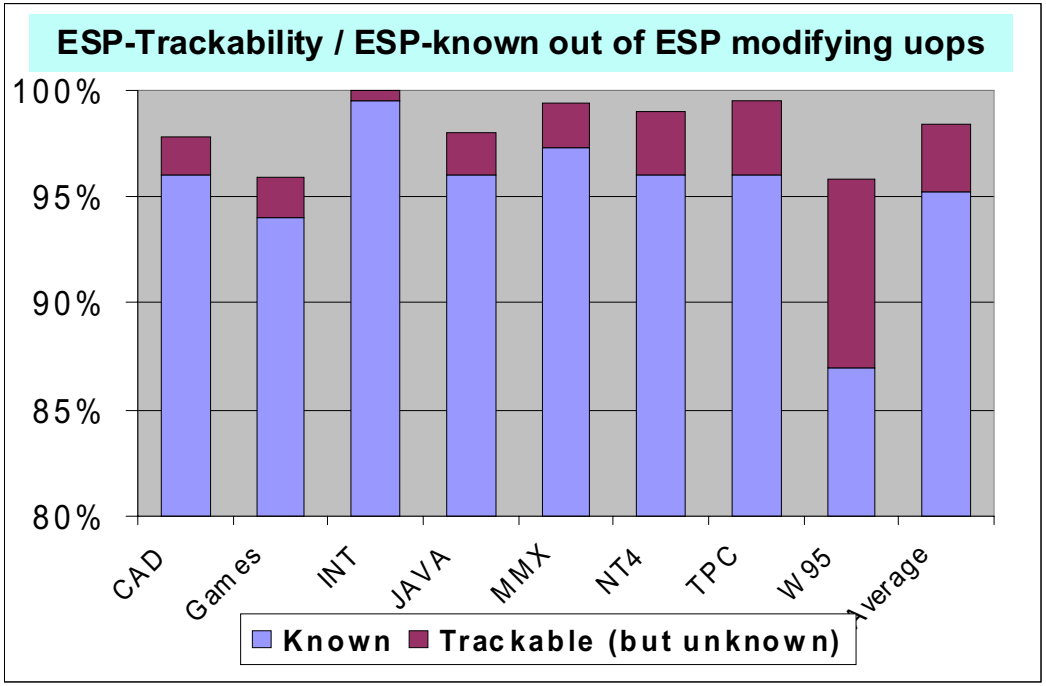
Assume 2-way decoder; D/R/S/E = Decode, Rename, Schedule, Execute

E - Execution w/o Stack Tracking; E - Execution with Stack Tracking;

intel

ESP Trackability

- When an untrackable ESP-modifying uop is decoded the ESP **value** is unknown until that uop is executed

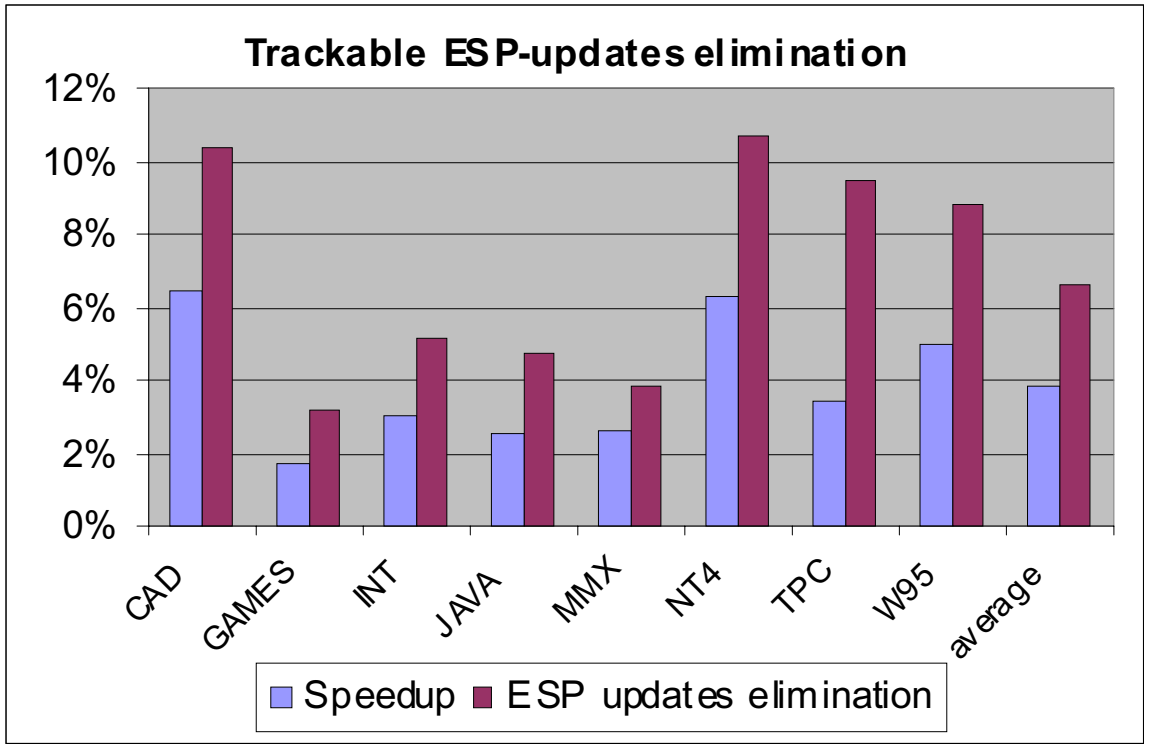


- 6-wide machine
- **8 pipe stages from decode to execution**
- 128-entry instruction window
- 4 integer, 1 complex, 2 ld/st and 1 FP units
- 64KB L1 data cache, 4-cycle hit-latency
- 1MB L2 cache size, 14-cycle hit-latency
- Hybrid Branch Predictor

- Relatively low trackability in Windows 95 due to
  - Large number of switches between privilege levels
    - change the stack (segment and ESP) in an untrackable way
  - Different coding style

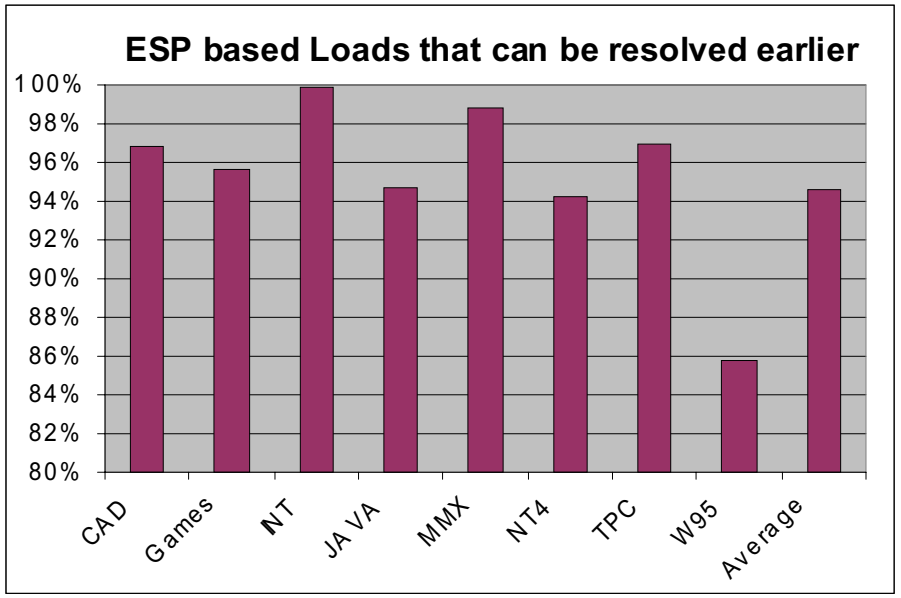
Cutting Dependencies / Reducing Uops

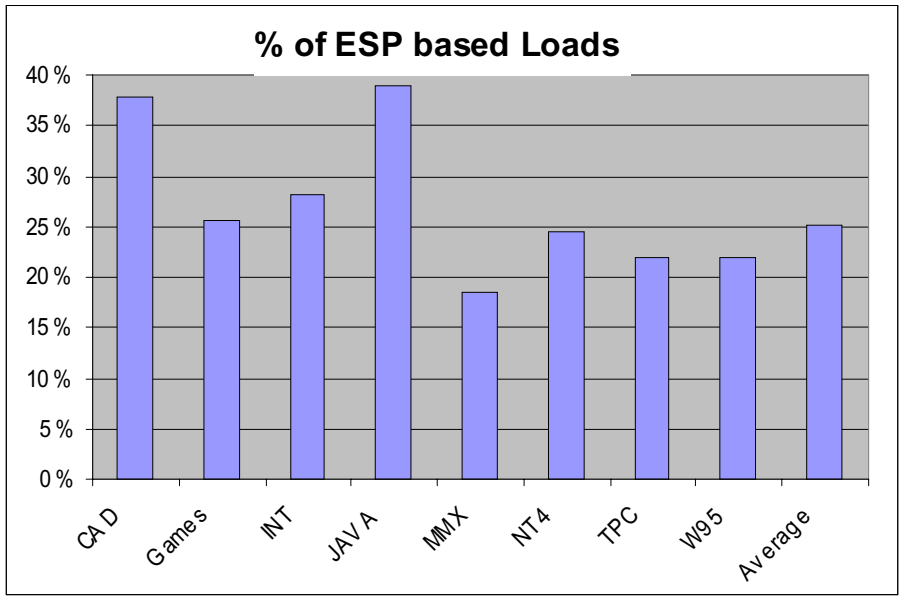
- Most *Trackable* ESP-updating uops can be "executed" in the FE (potentially saving ROB entries and execution BW)
- ➡ Potentially 7% of the uops (trackable) can be eliminated
- Potential performance gain of ESP updates elimination (on average) is ~4%. Coming from:
  - Dependency elimination
  - Bandwidth reduction

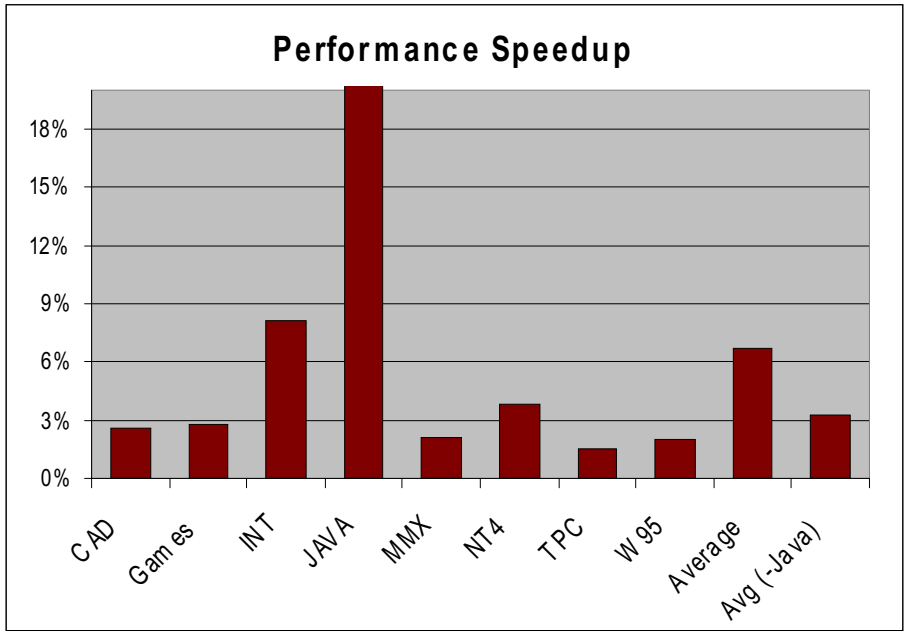


Early Stack Load address resolution

- 25% of all memory loads access the stack: **e.g.** **LD EAX, [ESP+4];**
- 95% of these references can be early calculated
- Potential Speedup: 6% (3% w/o Java)





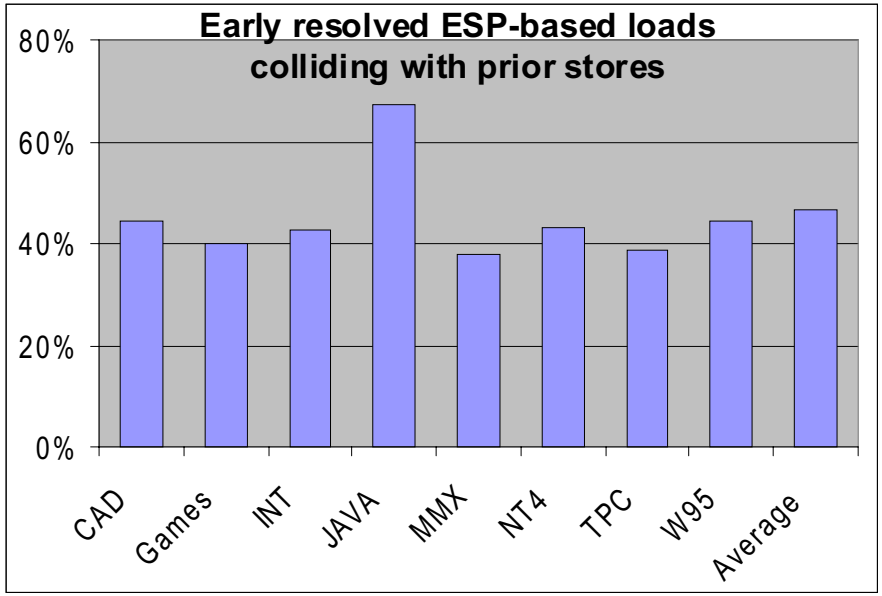


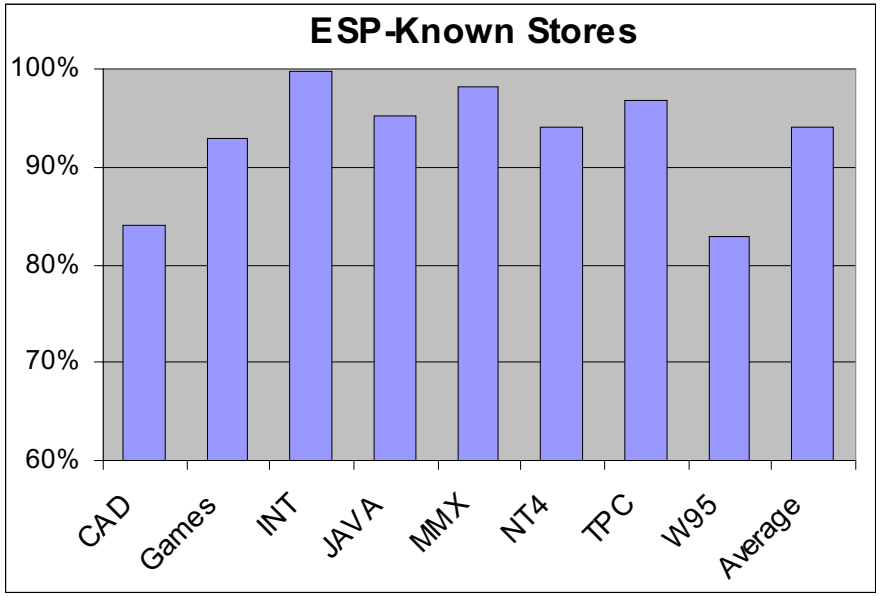
Memory Dependencies

- Early load fetches Ł more memory-order violations
  - The number of potentially bypassed stores increases
  - ESP-based loads are more likely to collide w/ prior stores
- Solution #1: use the tracking mechanism to track store addresses
  - J Safely determines whether a load can bypass prior trackable stores. But...
  - L Untrackable store address blocks succeeding loads from bypassing
  - ➨ Simple, but limited...
- Solution #2: use also memory dependency predictors to determine when early loads can be advanced
  - J Better coverage.
  - L More collisions Ł more pressure on the disambiguation structures

Memory Dependencies (con't)

- Stack references manifest high # of collisions (45%!)
  - Reason: stack operations exhibiting short-term producer-consumer dependencies
- About 95% of the ESP-based stores are ESP-known



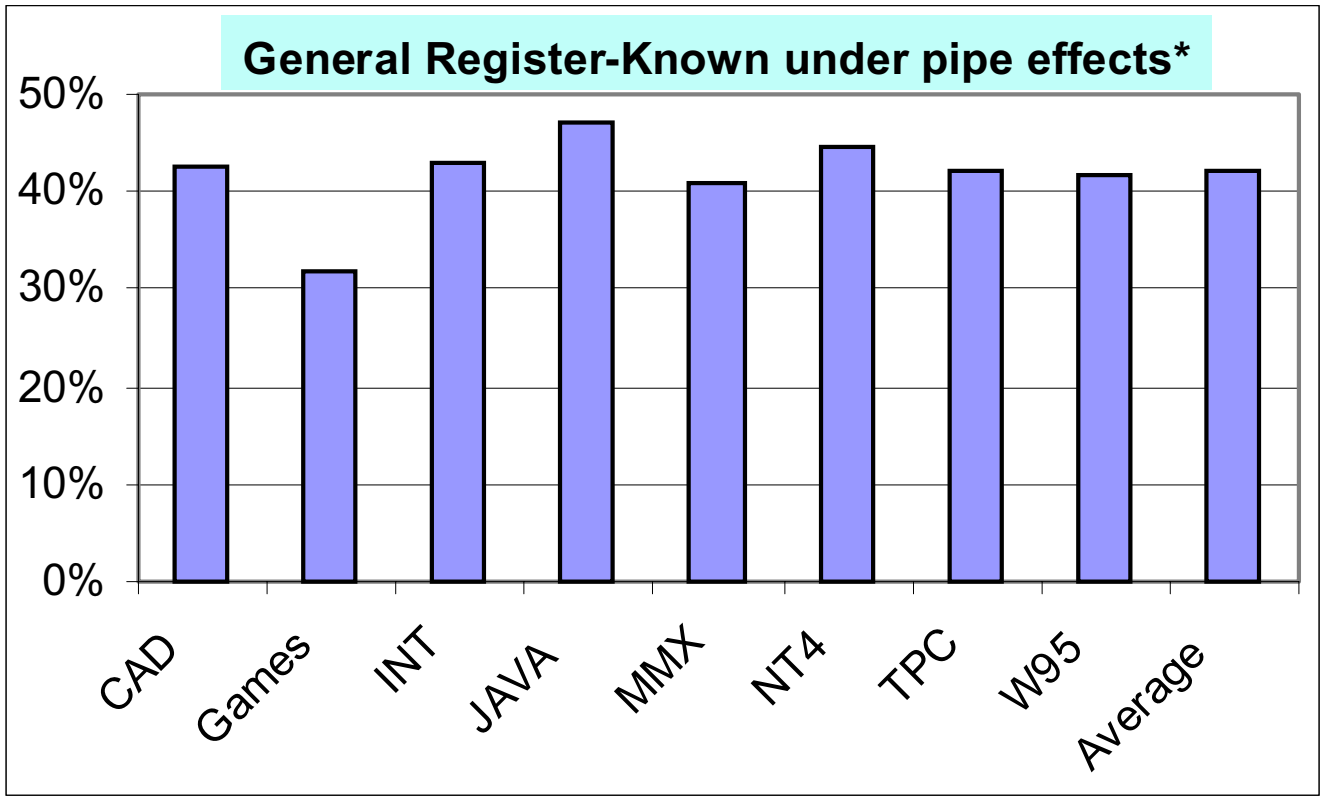


General Register Tracking

- General Purpose Registers are much less trackable

  Using similar mechanism used for ESP Tracking =>

  - *Only 40% of the modifications to GPRs are known, that is*
    - They are of the form $\boldsymbol{reg_2 = reg_1 \pm immediate}$
    - $\boldsymbol{reg_1}$ is known in the Front-End



* 8-stage pipeline from decode to execution

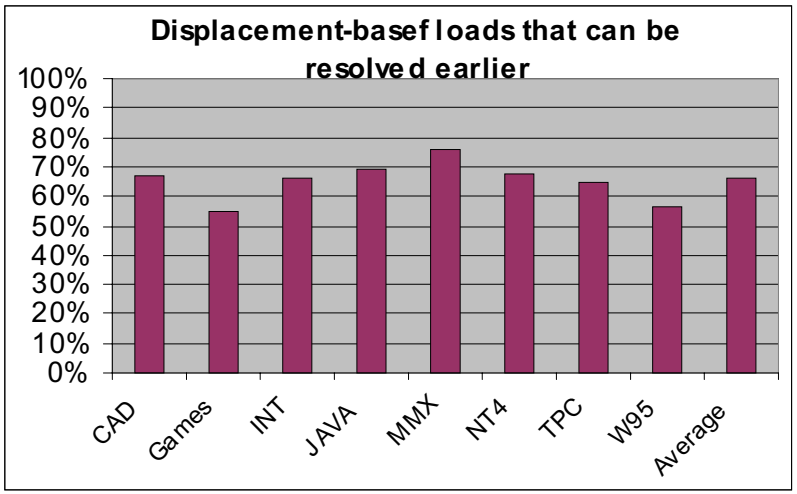
Early load address resolution

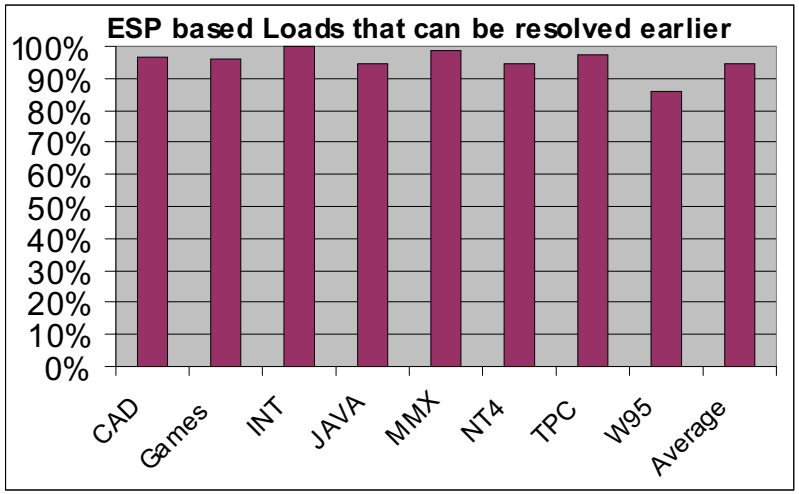
- Stack *address* - 25% of all memory loads access the stack
  - e.g. **LD EAX, [ESP+4];**
- *Absolute (constant) address* - 10% of loads
  - e.g. **LD EAX, [1000]**
- Displacement based loads - 82% of loads **(including stack)**
  - e.g. **LD EAX, [EBX+100]**

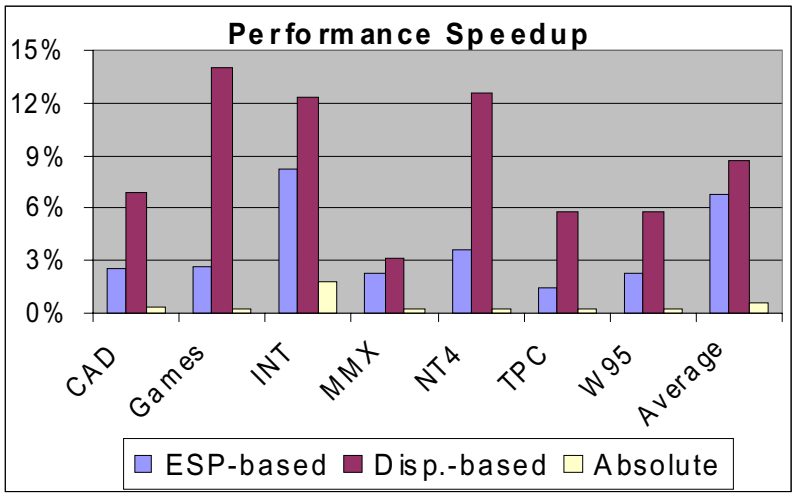
Benchmarks	Load reference type		
	ESP-based	Absolute	Disp.-based
CAD	38%	2%	88%
Games	26%	16%	77%
INT	28%	15%	77%
JAVA	39%	5%	89%
MMX	18%	13%	70%
NT4	24%	6%	88%
TPC	22%	9%	85%
W95	23%	7%	87%
Average	25%	10%	82%

Early address resolution (Cont.)

- Addresses of most stack references can be safely calculated
- Addresses of ~60% of the displacement-based loads can be early resolved





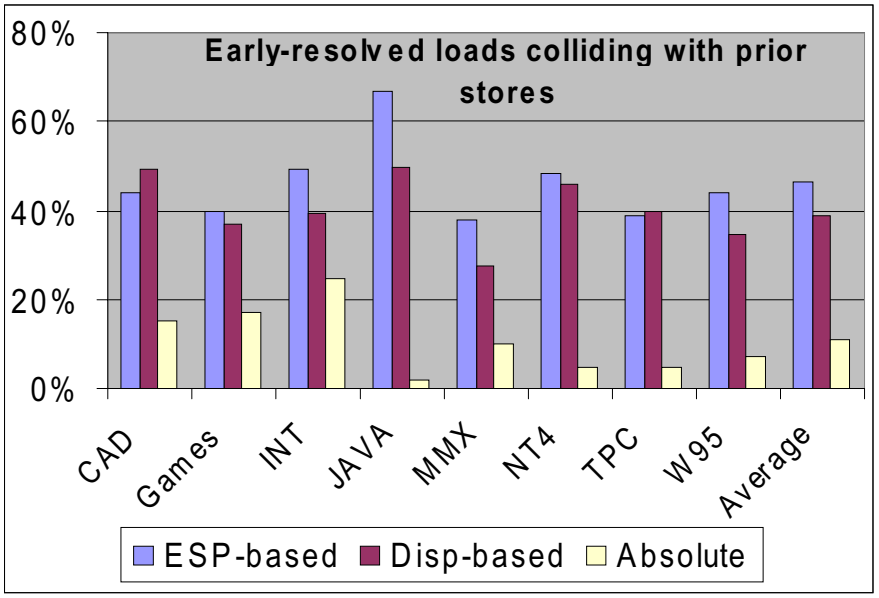


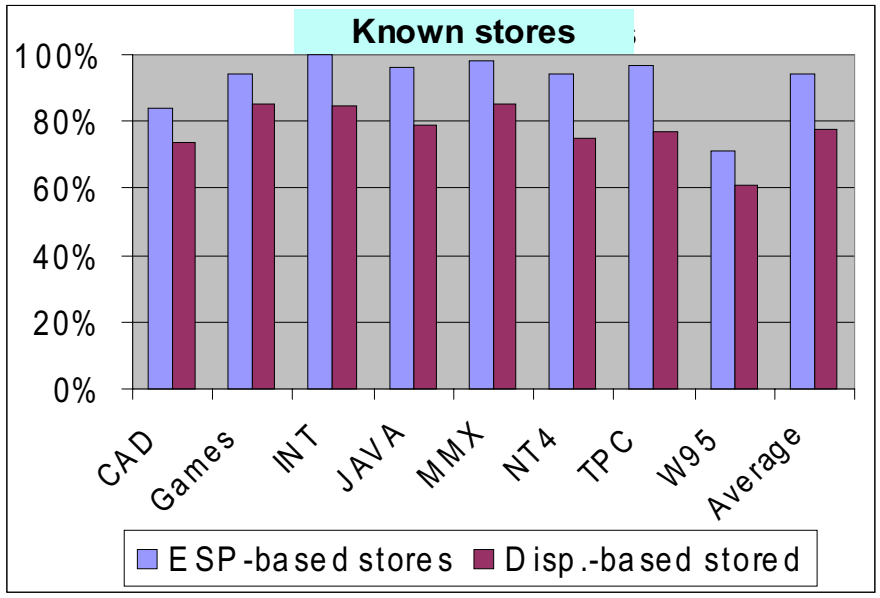
Memory Dependencies (General)

- General memory references tend to collide less than stack references (~40% vs 45%)

But...

- General stores are less trackable/known (<80% vs 95%)
  - General store tracking does not look practical





Conclusions

- Many memory loads reference stack - 25%
- ESP modifications are usually made of a simple *esp ± imm*
  - Highly trackable - 98%, highly known
- FE computation of ESP values reduces uop stream by ~7% and cuts dependencies Ł increases performance by ~4%
- Hiding the load-use delay via early resolution of ESP-based loads results in 6% speedup *(orthogonal to the above)*
- 95% of the ESP-based stores can be resolved earlier assisting the disambiguation mechanism
- Applying the tracking mechanism to all registers increases the potential gain but imposes a much greater complexity

Non-Speculative Technique